

3. Напишите программу на Java, которая принимает строку и переворачивает её без использования стандартных методов Java для переворота строк. Используйте цикл для замены символов местами.

Вот решение задачи на Java, где строка переворачивается вручную с помощью цикла и обмена символов, без использования стандартных методов вроде

```
import java.util.Scanner;

public class ReverseStringManual {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Введите строку: ");
        String input = scanner.nextLine();

        // Перевод строки в массив символов
        char[] chars = input.toCharArray();

        // Обмен символов местами с двух концов
        for (int i = 0; i < chars.length / 2; i++) {
            char temp = chars[i];
            chars[i] = chars[chars.length - 1 - i];
            chars[chars.length - 1 - i] = temp;
        }

        // Создание новой строки из массива
        String reversed = new String(chars);
        System.out.println("Перевернутая строка: " + reversed);
    }
}
```

❖ В этом коде:

- Используется `char[]` для прямого доступа к символам строки.

- С помощью цикла и временной переменной temp символы меняются местами от концов к центру.

3. Напишите метод equals, который принимает в качестве параметров два стека целых чисел, метод возвращает true, если два стека равны, и в противном случае возвращает false. Чтобы считаться равными, два стека должны хранить одинаковую последовательность целочисленных значений в одном и том же порядке. Ваш метод заключается в проверке двух стеков, но перед завершением работы метода необходимо вернуть их в исходное состояние. Вы можете использовать один стек в качестве вспомогательного хранения.

Вот пример реализации метода equals, который сравнивает два стека целых чисел и возвращает true, если они равны по содержимому и порядку, а также **восстанавливает исходное состояние стеков** после сравнения, используя **третий вспомогательный стек**:

java

КопироватьРедактировать

```
import java.util.Stack;
```

```
public class StackEquality {
```

```
    public static boolean equals(Stack<Integer> stack1,
Stack<Integer> stack2) {
```

```
        Stack<Integer> temp = new Stack<>();
```

```
        boolean areEqual = true;
```

```
        // Если размеры стеков разные — они точно не равны
```

```
        if (stack1.size() != stack2.size()) {
```

```
            return false;
```

```
}
```

```
        // Сравниваем элементы и временно перекладываем их в
temp
```

```
        while (!stack1.isEmpty()) {
```

```
            int val1 = stack1.pop();
```

```
            int val2 = stack2.pop();
```

```
    if (val1 != val2) {

        areEqual = false;

    }

    temp.push(val1); // можно класть только один —
достаточно, так как значения одинаковые

}

// Восстанавливаем исходное состояние стеков
while (!temp.isEmpty()) {

    int value = temp.pop();

    stack1.push(value);
    stack2.push(value);

}

return areEqual;
}

public static void main(String[] args) {

    Stack<Integer> s1 = new Stack<>();
    Stack<Integer> s2 = new Stack<>();

    s1.push(1);
    s1.push(2);
    s1.push(3);

    s2.push(1);
    s2.push(2);
    s2.push(3);
}
```

```
        System.out.println("Стэки равны? " + equals(s1, s2));
// true

        System.out.println("После вызова:");
        System.out.println("s1: " + s1);
        System.out.println("s2: " + s2);

    }

}
```

Объяснение:

- Сначала сравниваются размеры.
- Затем оба стека поэлементно распаковываются и сравниваются.
- Все элементы временно сохраняются во вспомогательном стеке temp.
- После сравнения — стек восстанавливается из temp.

Такой метод аккуратен, не разрушает данные и работает корректно для любых целочисленных стеков.

3. Создайте абстрактный класс Animal, который будет содержать абстрактный метод speak(). Далее создайте несколько классов-наследников:

Cow, который будет переопределять метод speak() и выводить "Moo!".

Cat, который будет переопределять метод speak() и выводить "Meow!".

Создайте интерфейс Domesticable, который будет требовать метод getHome().

Реализуйте его для классов Dog и Cat. Дополнительно нарисуйте диаграмму классов для данной задачи с использованием UML.

Вот как можно реализовать эту задачу на Java, включая **абстрактный класс Animal**, его наследников, интерфейс Domesticable, и UML-диаграмму:

Java-код:

◆ Абстрактный класс Animal:

java

Копировать Редактировать

```
public abstract class Animal {
```

```
    public abstract void speak();
```

```
}
```

◆ **Интерфейс Domesticable:**

java

КопироватьРедактировать

```
public interface Domesticable {
```

```
    void getHome();
```

```
}
```

◆ **Класс Dog:**

java

КопироватьРедактировать

```
public class Dog extends Animal implements Domesticable {
```

```
    @Override
```

```
    public void speak() {
```

```
        System.out.println("Woof!");
```

```
}
```

```
    @Override
```

```
    public void getHome() {
```

```
        System.out.println("Dog returns to the kennel.");
```

```
}
```

```
}
```

◆ **Класс Cat:**

java

КопироватьРедактировать

```
public class Cat extends Animal implements Domesticable {
```

```
    @Override
```

```
    public void speak() {
```

```
        System.out.println("Meow!");
```

```
}
```

```
@Override  
public void getHome() {  
    System.out.println("Cat jumps on the windowsill.");  
}  
}
```

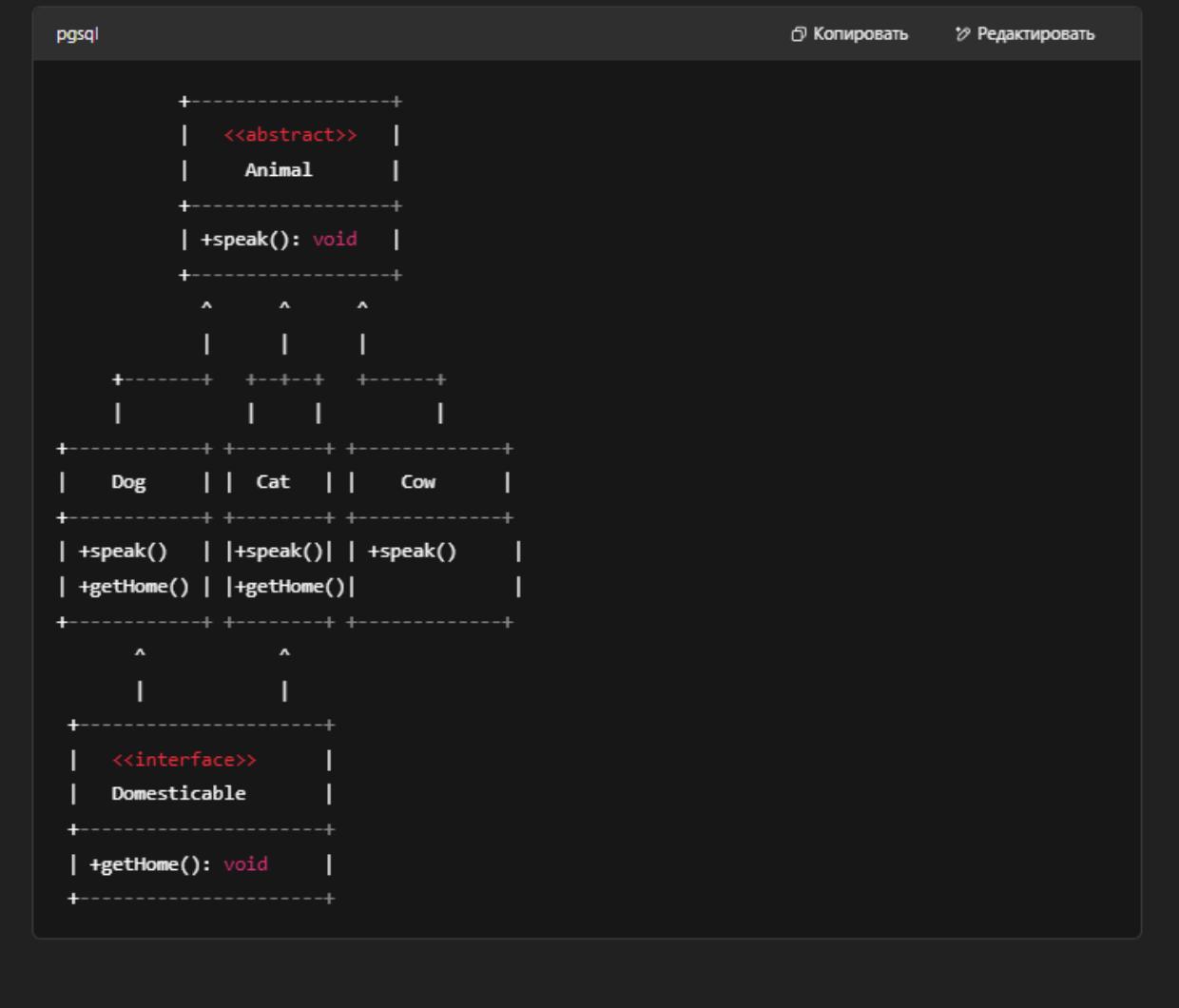
◆ Класс Cow:

java

КопироватьРедактировать

```
public class Cow extends Animal {  
    @Override  
    public void speak() {  
        System.out.println("Moo!");  
    }  
}
```

UML-диаграмма (текстовая версия):



3. Напишите метод `splitStack`, который принимает стек целых чисел в качестве параметра и разбивает его на отрицательные и неотрицательные значения. Числа в стеке должны быть переставлены так, чтобы все отрицательные значения появлялись в нижней части стека, а все неотрицательные - в верхней части. Другими словами, если после вызова этого метода вам нужно будет вытолкнуть числа из стека, вы сначала получите все неотрицательные числа, а затем получите все отрицательные числа. Неважно, в каком порядке появляются числа, если все отрицательные находятся в стеке всегда ниже, чем все неотрицательные числа. Вы можете использовать одну очередь в качестве вспомогательного хранения.

Отличная задача на работу со стеком и очередью!

Вот реализация метода `splitStack`, который делает именно то, что требуется:

Java-код решения:

```
java
```

[Копировать](#)[Редактировать](#)

```
import java.util.*;  
  
public class StackSplitter {  
    public static void splitStack(Stack<Integer> stack) {  
        Queue<Integer> queue = new LinkedList<>();  
  
        // Шаг 1: выгружаем всё из стека в очередь  
        while (!stack.isEmpty()) {  
            queue.add(stack.pop());  
        }  
  
        // Шаг 2: сначала добавляем отрицательные в стек  
        int size = queue.size();  
        for (int i = 0; i < size; i++) {  
            int num = queue.remove();  
            if (num < 0) {  
                stack.push(num);  
            } else {  
                queue.add(num); // снова в очередь — будем разбирать позже  
            }  
        }  
  
        // Шаг 3: теперь добавляем неотрицательные в стек (они будут сверху)  
        while (!queue.isEmpty()) {  
            stack.push(queue.remove());  
        }  
    }  
}
```

```

// Для проверки:

public static void main(String[] args) {

    Stack<Integer> numbers = new Stack<>();
    numbers.push(5);
    numbers.push(-3);
    numbers.push(7);
    numbers.push(0);
    numbers.push(-1);

    System.out.println("До: " + numbers);
    splitStack(numbers);
    System.out.println("После: " + numbers);

    // Демонстрация, что неотрицательные сверху
    while (!numbers.isEmpty()) {
        System.out.println(numbers.pop());
    }
}

```

Пояснение:

- Используется **только одна очередь** queue для временного хранения.
 - В стеке сначала оказываются **все отрицательные** (они внизу).
 - Затем — **все неотрицательные** (они вверху).
 - Порядок внутри групп **не важен**, как указано в условии.
- Создайте абстрактный класс Employee, содержащий общие для всех сотрудников поля: name, age, salary и абстрактный метод calculateBonus().
Далее создайте два класса-наследника:
 - Developer, который будет переопределять метод calculateBonus() и добавлять бонус в зависимости от количества завершённых проектов.

- Manager, который будет переопределять метод calculateBonus() и добавлять бонус в зависимости от количества подчинённых.
- Создайте интерфейс Reportable, который требует метод generateReport(). Реализуйте этот интерфейс только в классе Manager.
- Значение бонусов может быть любым, например, бонус 100 за каждого подчинённого и бонус 200 за каждый завершённый проект.

Вот полная реализация задачи с использованием **абстрактного класса, наследования и интерфейса**:

Абстрактный класс Employee:

java

КопироватьРедактировать

```
public abstract class Employee {  
    protected String name;  
    protected int age;  
    protected double salary;  
  
    public Employee(String name, int age, double salary) {  
        this.name = name;  
        this.age = age;  
        this.salary = salary;  
    }  
  
    public abstract double calculateBonus();  
  
    public void displayInfo() {  
        System.out.println("Имя: " + name);  
        System.out.println("Возраст: " + age);  
        System.out.println("Оклад: " + salary);  
        System.out.println("Бонус: " + calculateBonus());  
    }  
}
```

```
}
```

Интерфейс Reportable:

java

КопироватьРедактировать

```
public interface Reportable {
```

```
    void generateReport();
```

```
}
```

Класс Developer:

java

КопироватьРедактировать

```
public class Developer extends Employee {
```

```
    private int completedProjects;
```

```
    public Developer(String name, int age, double salary, int completedProjects) {
```

```
        super(name, age, salary);
```

```
        this.completedProjects = completedProjects;
```

```
}
```

```
@Override
```

```
    public double calculateBonus() {
```

```
        return 200 * completedProjects;
```

```
}
```

```
}
```

Класс Manager:

java

КопироватьРедактировать

```
public class Manager extends Employee implements Reportable {  
    private int numberOfSubordinates;  
  
    public Manager(String name, int age, double salary, int numberOfSubordinates) {  
        super(name, age, salary);  
        this.numberOfSubordinates = numberOfSubordinates;  
    }  
  
    @Override  
    public double calculateBonus() {  
        return 100 * numberOfSubordinates;  
    }  
  
    @Override  
    public void generateReport() {  
        System.out.println("Генерируется отчёт для менеджера: " + name);  
        System.out.println("Количество подчинённых: " + numberOfSubordinates);  
        System.out.println("Общий бонус: " + calculateBonus());  
    }  
}
```

 **Пример использования:**

```
java  
КопироватьРедактировать  
public class Main {  
    public static void main(String[] args) {  
        Developer dev = new Developer("Андрей", 28, 120000, 4);  
        Manager mgr = new Manager("Екатерина", 35, 150000, 7);
```

```

    dev.displayInfo();

    System.out.println("-----");

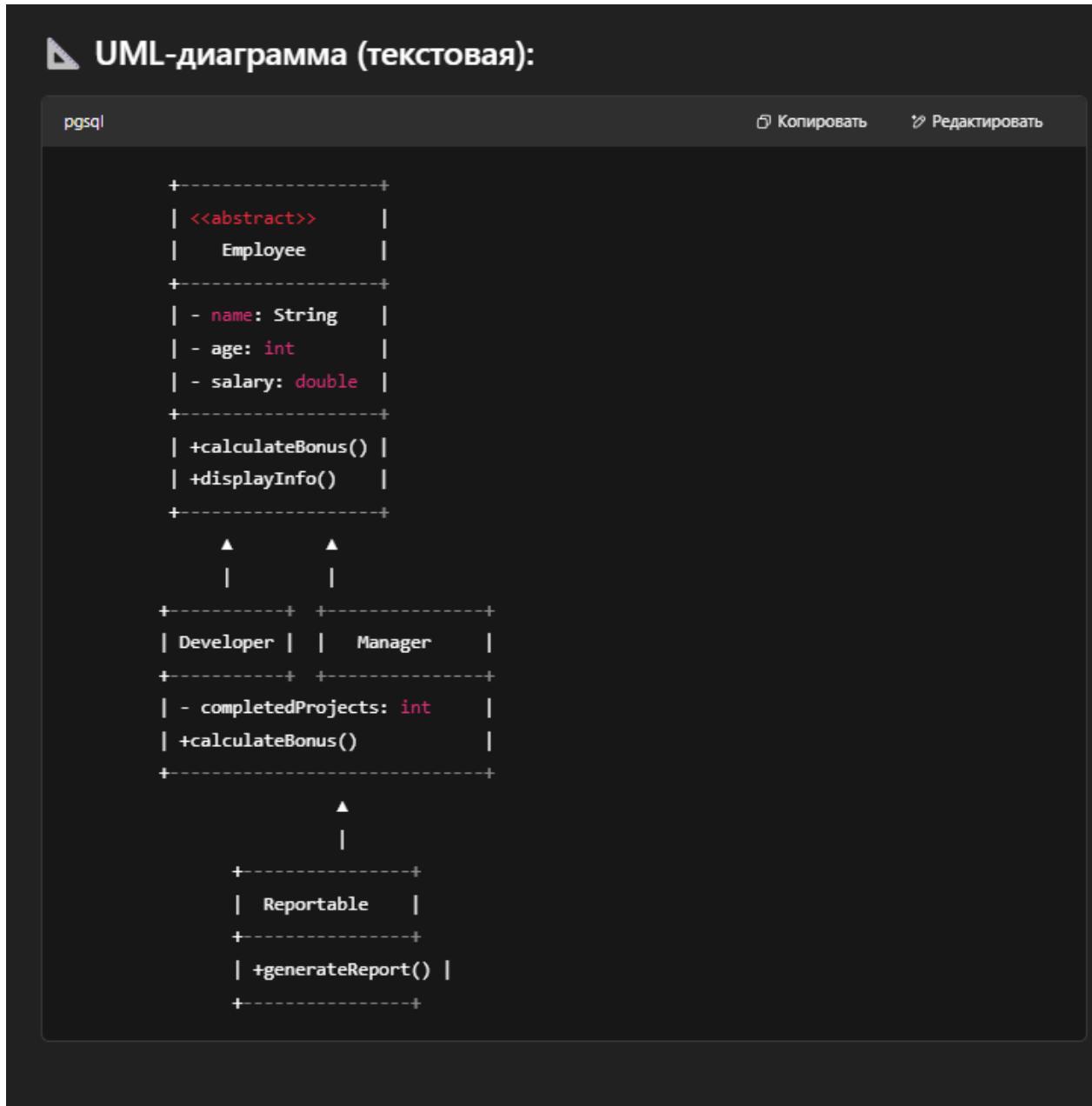
    mgr.displayInfo();

    mgr.generateReport();

}

}

```



Напишите метод под названием `alternate`, который принимает два списка целых чисел в качестве параметров и возвращает новый список, содержащий чередующиеся элементы из двух списков, в следующем порядке:

- Первый элемент из первого списка

- Первый элемент из второго списка

- Второй элемент из первого списка

- Второй элемент из второго списка

- Третий элемент из первого списка

- Третий элемент из второго списка

Если списки не содержат одинаковое количество элементов, оставшиеся элементы из более длинного списка должны быть расположены последовательно в конце.

Например, для первого списка (1, 2, 3, 4, 5) и второго списка (6, 7, 8, 9, 10, 11, 12) вызов alternate (list1, list2) должен вернуть список, содержащий (1, 6, 2, 7, 3, 8, 4, 9, 5, 10, 11, 12). Не изменяйте передаваемые списки параметров.

 **Java-код:**

java

Копировать Редактировать

```
import java.util.*;
```

```
public class ListAlternator {
```

```
    public static List<Integer> alternate(List<Integer> list1, List<Integer> list2) {
```

```
        List<Integer> result = new ArrayList<>();
```

```
        int size1 = list1.size();
```

```
        int size2 = list2.size();
```

```
        int minSize = Math.min(size1, size2);
```

```
        // Чередуем элементы до минимального размера
```

```
        for (int i = 0; i < minSize; i++) {
```

```
            result.add(list1.get(i));
```

```
            result.add(list2.get(i));
```

```
}
```

```

// Добавляем оставшиеся элементы из list1 (если есть)
for (int i = minSize; i < size1; i++) {
    result.add(list1.get(i));
}

// Добавляем оставшиеся элементы из list2 (если есть)
for (int i = minSize; i < size2; i++) {
    result.add(list2.get(i));
}

return result;
}

// Пример использования
public static void main(String[] args) {
    List<Integer> list1 = Arrays.asList(1, 2, 3, 4, 5);
    List<Integer> list2 = Arrays.asList(6, 7, 8, 9, 10, 11, 12);

    List<Integer> combined = alternate(list1, list2);
    System.out.println(combined); // [1, 6, 2, 7, 3, 8, 4, 9, 5, 10, 11, 12]
}

```

Особенности:

- Не изменяет оригинальные списки (list1 и list2).
- Работает с любыми размерами списков.
- Возвращает **новый список**.

Необходимо реализовать функцию, которая «нормализует» указанный относительный путь в файловой системе. Путь передаётся строкой, в которой используются следующие обозначения:

текущая директория;

родительская директория относительно текущей;

разделитель директорий.

Функция должна исключить лишние элементы пути, учитывая переходы с использованием ":" и вернуть нормализованный путь.

Пример:

Вход:

"MIREA/JAVA_5_2/src//.../.task_5_1/.../.../course/prog"

Выход:

".../course/prog"

Замечание: необходимо обратить внимание на структуру примера и вспомнить структуру файловой системы.

Java-код (метод normalizePath):

java

КопироватьРедактировать

import java.util.*;

public class PathNormalizer {

 public static String normalizePath(String path) {

 String[] parts = path.split("/");

 Deque<String> stack = new ArrayDeque<>();

 for (String part : parts) {

 if (part.equals(".")) || part.isEmpty()) {

 // Пропускаем текущую директорию и пустые сегменты

```
        continue;

    } else if (part.equals(".")) {

        // Возврат назад — удаляем верхний элемент, если есть, иначе добавляем "."

        if (!stack.isEmpty() && !stack.peek().equals(".")) {

            stack.pop();

        } else {

            stack.push(".");

        }

    } else {

        stack.push(part);

    }

}

// Путь собран в стеке в обратном порядке — переворачиваем

List<String> normalized = new ArrayList<>();

while (!stack.isEmpty()) {

    normalized.add(stack.removeLast());

}

return String.join("/", normalized);

}

// Пример

public static void main(String[] args) {

    String input = "MIREA/JAVA_5_2/src/../../task_5_1/../../course/prog";

    String output = normalizePath(input);

    System.out.println("Нормализованный путь: " + output); // ../course/prog

}
```

 **Пояснение:**

- . игнорируется.
- .. удаляет последний добавленный путь, если он есть.
- Используем стек (Deque) для отслеживания пути.
- В конце переворачиваем стек для корректного порядка директорий.

3. Напишите метод `isPalindrome`, который принимает в качестве параметра очередь целых чисел и возвращает `true`, если числа в очереди представляют палиндром (и `false` в противном случае). Последовательность чисел считается палиндромом, если она совпадает в обратном порядке. Например, предположим, что очередь с именем `q` хранит эти значения:

`front [3, 8, 17, 9, 17, 8, 3] back`

Тогда вызов `isPalindrome (q);` должен вернуть `true`, потому что эта последовательность одинакова в обратном порядке. Если в очереди хранятся эти значения:

`front [3, 8, 17, 9, 4, 17, 8, 3] back`

Вызов `isPalindrome` вместо этого вернул бы `false`, потому что эта последовательность не совпадает обратном порядке (9 и 4 в середине не совпадают). Пустую очередь следует считать палиндромом. Вы можете делать какие-либо предположения относительно количества элементов в очереди, и ваш метод должен восстановить очередь, после использования очереди, чтобы после вызова он сохранял ту последовательность значений, что и раньше. Вы можете использовать стек в качестве вспомогательн хранения.

Вот реализация метода `isPalindrome`, который проверяет, является ли очередь палиндромом, **не нарушая порядок элементов**, и используя стек как вспомогательную структуру:

 **Java-код:**

`java`

[Копировать](#)[Редактировать](#)

```
import java.util.*;
```

```
public class PalindromeChecker {  
  
    public static boolean isPalindrome(Queue<Integer> queue) {  
        Stack<Integer> stack = new Stack<>();  
  
        int size = queue.size();  
        boolean isPalin = true;  
  
        // Сохраняем элементы в стек и обратно в очередь (чтобы не потерять порядок)  
        for (int i = 0; i < size; i++) {  
            int val = queue.remove();  
            stack.push(val);  
            queue.add(val); // возвращаем в очередь  
        }  
  
        // Сравниваем элементы из очереди и стека (L → R и R → L)  
        for (int i = 0; i < size; i++) {  
            int val = queue.remove();  
            int top = stack.pop();  
            if (val != top) {  
                isPalin = false;  
            }  
            queue.add(val); // снова возвращаем в очередь  
        }  
  
        return isPalin;  
    }  
  
    // Пример использования  
    public static void main(String[] args) {
```

```
Queue<Integer> q1 = new LinkedList<>(Arrays.asList(3, 8, 17, 9, 17, 8, 3));  
Queue<Integer> q2 = new LinkedList<>(Arrays.asList(3, 8, 17, 9, 4, 8, 3));  
  
System.out.println(isPalindrome(q1)); // true  
System.out.println(isPalindrome(q2)); // false  
}  
}
```

Особенности:

- Пустая очередь считается палиндромом.
- Порядок в очереди **сохраняется** после проверки.
- Используется **только один стек** как вспомогательное хранилище.

3. Напишите программу, которая открывает файл для чтения, выполняет операцию с содержимым и всегда закрывает файл в блоке finally, даже если возникло исключение при чтении. Если файл не существует, выбрасывается исключение FileNotFoundException. В случае других ошибок (например, ошибки чтения) выбрасывается IOException. Необходимо гарантировать, что файл будет закрыт в любом случае. Расскажите возможно ли использование каких-то иных конструкций, кроме блока finally для решения данной задачи?

Вот пример Java-программы, которая соответствует описанию — **гарантирует закрытие файла** в блоке finally, даже если возникли исключения:

Пример с finally:

```
java
```

КопироватьРедактировать

```
import java.io.*;
```

```
public class FileReaderExample {  
    public static void main(String[] args) {
```

```
BufferedReader reader = null;

try {
    reader = new BufferedReader(new FileReader("data.txt"));

    String line;
    while ((line = reader.readLine()) != null) {
        System.out.println(line); // любая операция с содержимым
    }
} catch (FileNotFoundException e) {
    System.err.println("Файл не найден: " + e.getMessage());
} catch (IOException e) {
    System.err.println("Ошибка чтения файла: " + e.getMessage());
} finally {
    // Гарантированное закрытие файла
    if (reader != null) {
        try {
            reader.close();
        } catch (IOException e) {
            System.err.println("Ошибка при закрытии файла: " + e.getMessage());
        }
    }
}
```

Альтернатива finally: конструкция try-with-resources

Да, начиная с **Java 7**, появилась **альтернативная конструкция** — `try-with-resources`, которая **автоматически закрывает** ресурсы (файлы, сокеты и т.п.), **без необходимости писать finally**.

Пример:

```
java
```

[Копировать](#)[Редактировать](#)

```
import java.io.*;  
  
public class FileReaderWithResources {  
    public static void main(String[] args) {  
        try (BufferedReader reader = new BufferedReader(new FileReader("data.txt"))){  
            String line;  
            while ((line = reader.readLine()) != null){  
                System.out.println(line);  
            }  
        } catch (FileNotFoundException e){  
            System.err.println("Файл не найден: " + e.getMessage());  
        } catch (IOException e){  
            System.err.println("Ошибка чтения файла: " + e.getMessage());  
        }  
    }  
}
```

3. Напишите обобщённый метод swap(), который меняет местами два объекта в массиве. Метод должен работать с любыми типами данных, используя дженерики.

Вот пример обобщённого (generic) метода swap(), который меняет местами два элемента в массиве **любого типа**:

Обобщённый метод swap в Java:

java

[Копировать](#)[Редактировать](#)

```
public class SwapUtil {
```

```
    public static <T> void swap(T[] array, int i, int j) {
```

```

if (array == null || i < 0 || j < 0 || i >= array.length || j >= array.length) {
    throw new IndexOutOfBoundsException("Некорректные индексы или массив.");
}

T temp = array[i];
array[i] = array[j];
array[j] = temp;
}

// Пример использования
public static void main(String[] args) {
    String[] words = {"apple", "banana", "cherry"};
    Integer[] numbers = {1, 2, 3, 4};

    swap(words, 0, 2); // меняем "apple" и "cherry"
    swap(numbers, 1, 3); // меняем 2 и 4

    System.out.println("Слова: " + java.util.Arrays.toString(words));
    System.out.println("Числа: " + java.util.Arrays.toString(numbers));
}
}

```

Особенности:

- Т — обобщённый тип (generic).
- Работает с массивами любого объектного типа (T[]), например String[], Integer[], Double[], CustomClass[].
- Для примитивов (int[], double[]) нужно писать отдельную реализацию или использовать обёртки (Integer[], Double[]).

3. Число Рамануджана - это такое число, которое может быть представлено как сумма двух кубов двумя различными способами. Одним из известных чисел Рамануджана является число 1729. Оно может быть представлено как сумма кубов двух чисел двумя способами:

$$1729 = 13 + 123$$

$$1729 = 93 + 103$$

Создайте программу, которая находит первое число Рамануджана, которое может быть представлено как сумма двух кубов двумя различными способами. Программа должна выводить это число и все пары чисел, которые приводят к его представлению как сумма кубов.

Условия задачи:

Используйте метод для поиска чисел Рамануджана.

Программа должна искать такие числа начиная с 1 и проверять возможные комбинации чисел, которые в сумме дадут одинаковое значение при возведении в куб.

Программа должна находить первое число Рамануджана и выводить его на экран.

Выведите все возможные пары чисел, чьи кубы дают это число.

Пример работы:

Число Рамануджана: 1729

Сумма кубов: $13 + 123 = 1729$

Сумма кубов: $93 + 103 = 1729$

Подсказка: Для поиска таких чисел используйте два вложенных цикла для перебора всех возможных чисел и их кубов, чтобы найти все пары, сумма кубов которых одинаковая. Рекомендуется использовать HashMap для хранения суммы кубов чисел и соответствующих пар чисел, которые дают эту сумму. Результаты работы должны соответствовать A001235 в OEIS.

Java-решение:

java

[Копировать](#)[Редактировать](#)

```
import java.util.*;
```

```
public class RamanujanNumber {
```

```
public static void main(String[] args) {  
    int limit = 100; // ограничение на перебор (можно увеличить)  
    Map<Integer, List<String>> cubeSums = new HashMap<>();  
  
    for (int a = 1; a <= limit; a++) {  
        for (int b = a; b <= limit; b++) { // b начинается с a, чтобы не было повторов  
            int sum = cube(a) + cube(b);  
            String pair = a + "^3 + " + b + "^3";  
  
            // Если такая сумма уже была — добавляем ещё одну пару  
            if (!cubeSums.containsKey(sum)) {  
                cubeSums.put(sum, new ArrayList<>());  
            }  
            cubeSums.get(sum).add(pair);  
  
            // Если нашли два разных способа получить эту сумму  
            if (cubeSums.get(sum).size() == 2) {  
                System.out.println("Число Рамануджана: " + sum);  
                System.out.println("Сумма кубов: " + cubeSums.get(sum).get(0) + " = " + sum);  
                System.out.println("Сумма кубов: " + cubeSums.get(sum).get(1) + " = " + sum);  
                return;  
            }  
        }  
    }  
  
    private static int cube(int n) {  
        return n * n * n;
```

}

}